43 | 虚拟机内核：KVM是什么？

2021-08-16 LMOS

《操作系统实战45讲》 课程介绍 >

讲述：陈晨

时长 12:40 大小 11.62M

你好，我是 LMOS。

上节课，我们理解了 Linux 里要如何实现系统 API。可是随着云计算、大数据和分布式技术的演进，我们需要在一台服务器上虚拟化出更多虚拟机，还要让这些虚拟机能够弹性伸缩，实现跨主机的迁移。

而虚拟化技术正是这些能力的基石。这一节课，就让我们一起探索一下，亚马逊、阿里、腾讯等知名公司用到的云虚拟主机，看看其中的核心技术——KVM 虚拟化技术。

理解虚拟化的定义

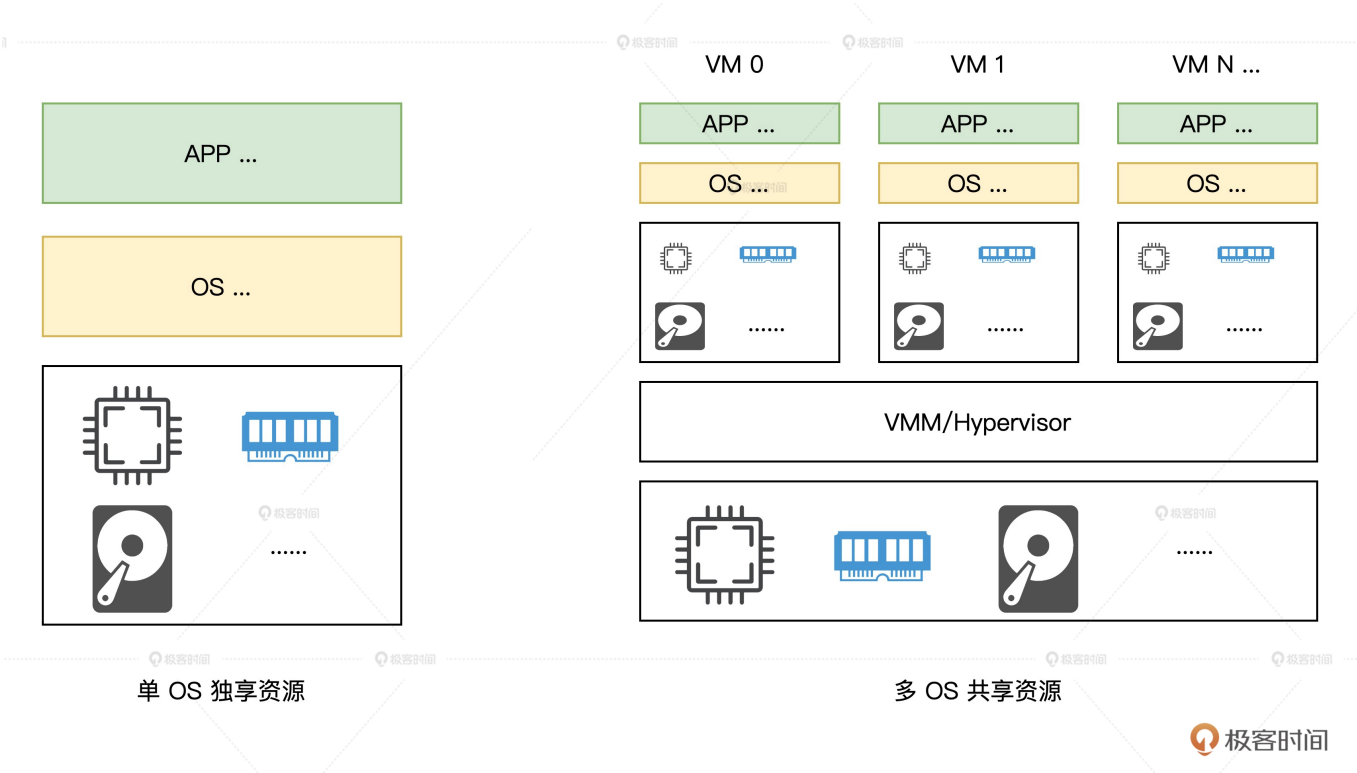
什么是虚拟化？在我看来，虚拟化的本质是一种资源管理的技术，它可以通过各种技术手段把计算机的实体资源（如：CPU、RAM、存储、网络、I/O 等等）进行**转换和抽象**，让这些资源可以重新分割、排列与组合，实现**最大化使用物理资源的目的**。

虚拟化的核心思想

学习了前面的课程我们发现，操作系统的设计很高明，已经帮我们实现了单机的资源配置需求，具体就是在一台物理机上把 CPU、内存资源抽象成进程，把磁盘等资源抽象出存储、文件、I/O 等特性，方便之后的资源调度和管理工作。

但随着时间的推移，我们做个统计就会发现，其实现在的 PC 机平常可能只有 50% 的时间处于工作状态，剩下的一半时间都是在闲置资源，甚至要被迫切换回低功耗状态。这显然是对资源的严重浪费，那么我们如何解决资源复用的问题呢？

这个问题确实很复杂，但根据我们的工程经验，但凡遇到不太好解决的问题，我们就可以考虑抽象出一个新的层次来解决。于是我们在已有的 OS 经验之上，进行了后面这样的设计。



虚拟化架构简图

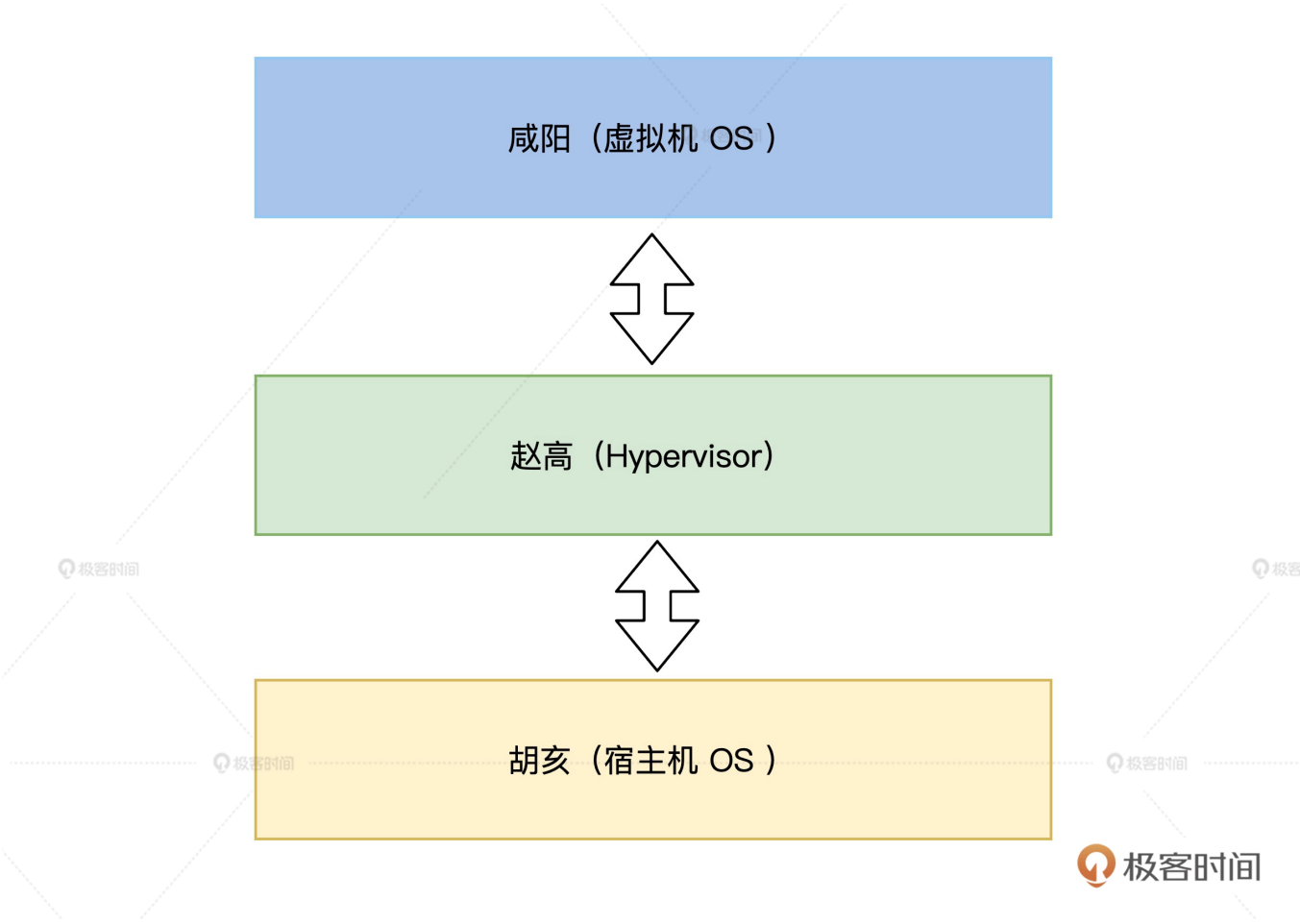
结合图解，可以看出最大的区别就是后者额外引入了一个叫 Hypervisor/Virtual Machine Monitor（VMM）的层。在这个层里面我们就可以做一些“无中生有”的事情，向下统一

管理和调度真实的物理资源，向上“骗”虚拟机，让每个虚拟机都以为自己都独享了独立的资源。

而在这个过程中，我们既然作为一个“两头骗的中间商”，显然要做一些瞒天过海的事情（访问资源的截获与重定向）。那么让我们先暂停两分钟，思考一下具体如何设计，才能实现这个“两头骗”的目标呢？

用赵高矫诏谈理解虚拟化

说起欺上瞒下，有个历史人物很有代表性，他就是赵高。始皇三十七年（前 210 年），统一了天下的秦始皇（OS）在生平最后一次出巡路上去世了，管理诏书的赵高（Hypervisor/VMM）却趁机发动了阴谋，威胁丞相李斯，矫诏处死扶苏与蒙恬。



赵高欺上瞒下

赵高隐瞒秦始皇死讯，还伪造了诏书，回到了咸阳最终一顿忽悠立了胡亥为为帝。这段故事后世称为沙丘之变。

作为一个成功瞒天过海，实现了偷梁换柱的中间人赵高，他成事的关键要点包括这些，首先要像咸阳方向伪造一切正常的假象（让被虚拟化的机器看起来和平常一样），其次还要把真正核心的权限获取到手（Hypervisor/VMM 要想办法调度真正的物理资源）。

所以以史为鉴。在具体实现的层面，我们会发现，这个瞒天过海的目标其实有几种实现方式。

一种思路是赵高一个人全权代理，全部模拟和代理出所有的资源（**软件虚拟化技术**），另一种思路是朝中有人（胡亥）配合赵高控制、调度各种资源的使用，真正执行的时候，再转发给胡亥去处理（**硬件虚拟化技术**）。

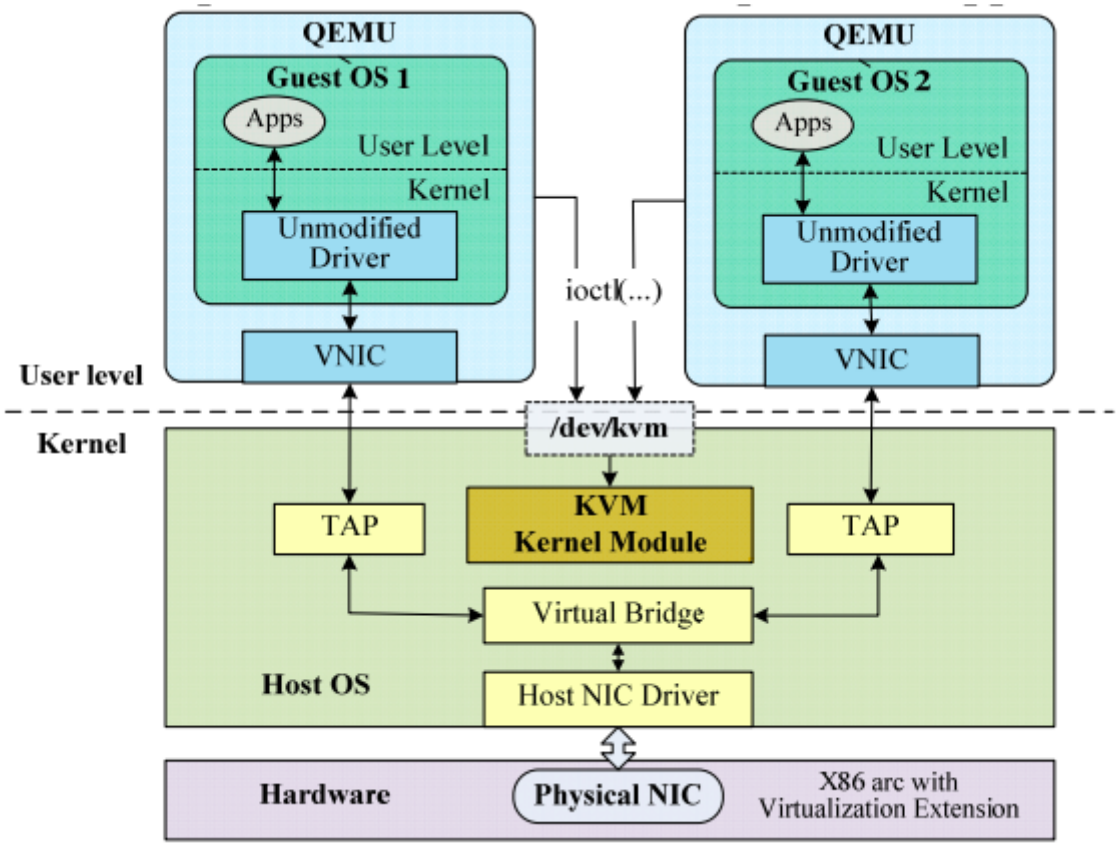
我们发现如果如果是前者，显然赵高会消耗大量资源，并且还可能会遇到一些安全问题，所以他选择了后者。

历史总是惊人地相似，在软件虚拟化遇到了无法根治的性能瓶颈和安全等问题的时候，软件工程师就开始给硬件工程师提需求了，需求背后的核心想法是这样的：能不能让朝中有人，有问题交给他，软件中间层只管调度资源之类的轻量级工作呢？

KVM 架构梳理

答案显然是可以的，根据我们对计算机的了解就会发现，计算机最重要几种资源分别是：计算（CPU）、存储（RAM、ROM），以及为了连接各种设备抽象出的 I/O 资源。

所以 Intel 分别设计出了 VT-x 指令集、VT-d 指令集、VT-c 指令集等技术来实现硬件虚拟化，让 CPU 配合我们来实现这个目标，了解了核心思想之后，让我们来看一看 KVM 的架构图。（图片出自论文《Residency-Aware Virtual Machine Communication Optimization: Design Choices and Techniques]》）



是不是看起来比较复杂？别担心，我用大白话帮你梳理一下。

首先，客户机（咸阳）看到的硬件资源基本都是由 Hypervisor（赵高）模拟出来的。当客户机对模拟设备进行操作时，命令就会被截获并转发给实际设备 / 内核模块（胡亥）去处理。

通过这种架构设计 Hypervisor 层，最终实现了把一个客户机映射到宿主机 OS 系统的一个进程，而一个客户机的 vCPU 则映射到这个进程下的独立的线程中。同理，I/O 也可以映射到同一个线程组内的独立线程中。

这样，我们就可以基于物理机 OS 的进程等资源调度能力，实现不同虚拟机的权限限定、优先级管理等功能了。

KVM 核心原理

通过前面的知识，我们发现，要实现成功的虚拟化，核心是要对资源进行“欺上瞒下”。而对应到我们计算机内的最重要的资源，可以简单抽象成为三大类，分别是：CPU、内存、I/O。接下来，我们就来看看如何让这三大类资源做好虚拟化。

CPU 虚拟化原理

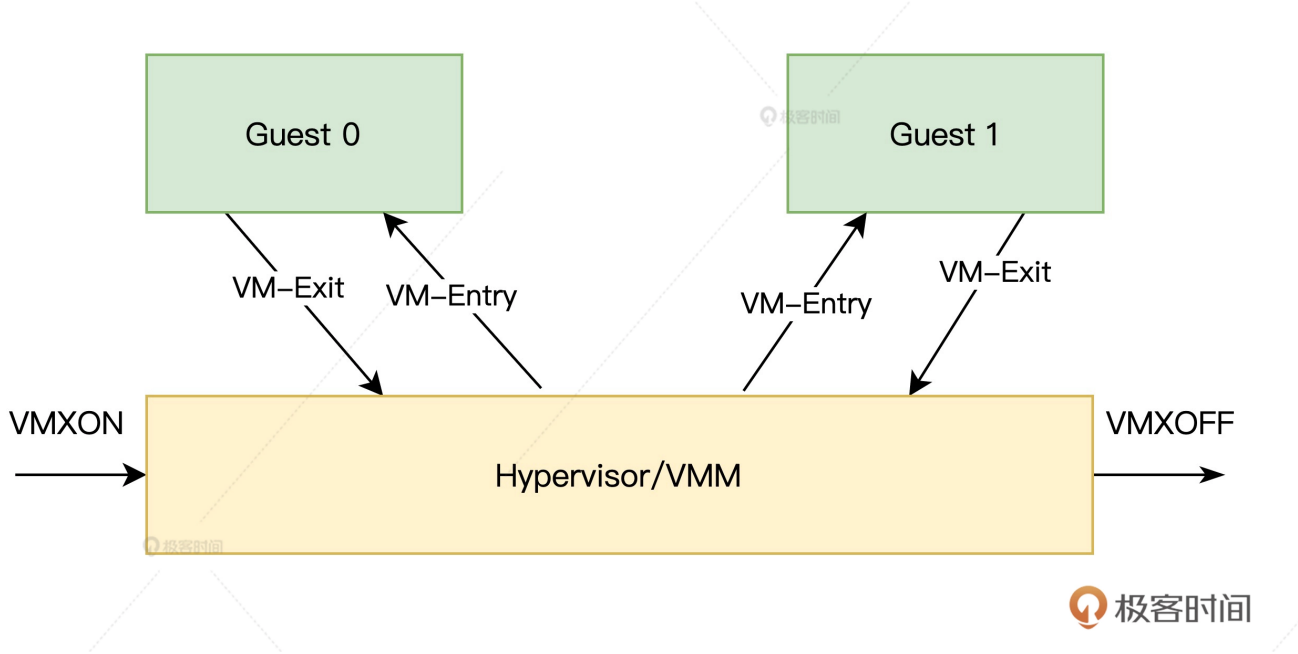
众所周知，CPU 是我们计算机最重要的模块，让我们先看看 Intel CPU 是如何跟 Hypervisor/VMM“里应外合”的。

Intel 定义了 Virtual Machine Extension（VMX）这个处理器特性，也就是传说中的 VT-x 指令集，开启了这个特性之后，就会存在两种操作模式。它们分别是：根操作（VMX root operation）和非根操作（VMX non-root operation）。

我们之前说的 Hypervisor/VMM，其实就运行在根操作模式下，这种模式下的系统对处理器和平台硬件具有完全的控制权限。

而客户软件（Guest software）包括虚拟机内的操作系统和应用程序，则运行在非根操作模式下。当客户软件执行一些特殊的敏感指令或者一些异常（如 CPUID、INVD、INVEPT 指令，中断、故障、或者一些寄存器操作等）时，则会触发 VM-Exit 指令切换回根操作模式，从而让 Hypervisor/VMM 完全接管控制权限。

下面这张图画出了模式切换的过程，想在这两种模式之间切换，就要通过 VM-Entry 和 VM-Exit 实现进入和退出。而在这个切换过程中，你要留意一个非常关键的数据结构，它就是 VMCS（Virtual Machine Control Structure）数据结构控制（下文也会讲到）。



VMM和Guest切换过程

内存虚拟化原理

内存虚拟化的核心目的是“骗”客户机，给每个虚拟客户机都提供一个从 0 开始的连续的物理内存空间的假象，同时又要保障各个虚拟机之间内存的隔离和调度能力。

可能有同学已经联想到，我们之前实现实现虚拟内存的时候，不也是在“骗”应用程序每个程序都有连续的物理内存，为此还设计了一大堆“转换表”的数据结构和转换、调度机制么？

没错，其实内存虚拟化也借鉴了相同的思想，只不过问题更复杂些，因为我们发现我们的内存从原先的虚拟地址、物理地址突然变成了后面这四种内存地址。

1. 客户机虚拟地址 GVA（Guest Virtual Address）

2. 客户机物理地址 GPA（Guest Physical Address）

3. 宿主机虚拟地址 HVA（Host Virtual Address）

4. 宿主机物理地址 HPA（Host Physical Address）

一看到有这么多种地址，又需要进行地址转换，想必转换时的**映射关系表**是少不掉的。

确实，早期我们主要是基于影子页表（Shadow Page Table）来进行转换的，缺点就是性能有不小的损耗。所以，后来 Intel 在硬件上就设计了 EPT（Extended Page Tables）机制，用来提升内存地址转换效率。

I/O 虚拟化原理

I/O 虚拟化是基于 Intel 的 VT-d 指令集来实现的，这是一种基于 North Bridge 北桥芯片（或 MCH）的硬件辅助虚拟化技术。

运用 VT-d 技术，虚拟机得以使用基于直接 I/O 设备分配方式，或者用 I/O 设备共享方式来代替传统的设备模拟 / 额外设备接口方式，不需要硬件改动，还省去了中间通道和 VMM 的开销，从而大大提升了虚拟化的 I/O 性能，让虚拟机性能更接近于真实主机。

KVM 关键代码走读

前面我们已经明白了 CPU、内存、I/O 这三类重要的资源是如何做到虚拟化的。不过知其然, 也要知其所以然，对知识只流于原理是不够的。接下来让我们来看看，具体到代码层面，虚拟化技术是如何实现的。

创建虚拟机

这里我想提醒你的是，后续代码为了方便阅读和理解，只保留了与核心逻辑相关的代码，省略了部分代码。

首先，我们来看一下虚拟机初始化的入口部分，代码如下所示。

复制代码

```
virt/kvm/kvm_main.c:
static int kvm_dev_ioctl_create_vm(void)
{
  int fd;
  struct kvm *kvm;

   kvm = kvm_create_vm(type);
   if (IS_ERR(kvm))
            return PTR_ERR(kvm);

   r = kvm_coalesced_mmio_init(kvm);

   r = get_unused_fd_flags(O_CLOEXEC);

         /*生成kvm-vm控制文件*/
   file = anon_inode_getfile("kvm-vm", &kvm_vm_fops, kvm, O_RDWR);

  return fd;
}
```

接下来。我们要创建 KVM 中内存、I/O 等资源相关的数据结构并进行初始化。

复制代码

```
virt/kvm/kvm_main.c:
static struct kvm *kvm_create_vm(void)
{
  int r, i;
    struct kvm *kvm = kvm_arch_create_vm();

    /*设置kvm的mm结构为当前进程的mm,然后引用计数为1*/
    kvm->mm = current->mm;
```

```
  kvm_eventfd_init(kvm);
  mutex_init(&kvm->lock);
  mutex_init(&kvm->irq_lock);
  mutex_init(&kvm->slots_lock);
  refcount_set(&kvm->users_count, 1);
  INIT_LIST_HEAD(&kvm->devices);
  INIT_HLIST_HEAD(&kvm->irq_ack_notifier_list);

  r = kvm_arch_init_vm(kvm, type);

  r = hardware_enable_all()

  for (i = 0; i < KVM_NR_BUSES; i++) {
    rcu_assign_pointer(kvm->buses[i],
  kzalloc(sizeof(struct kvm_io_bus), GFP_KERNEL));
  }
  kvm_init_mmu_notifier(kvm);

    /*把kvm链表加入总链表*/
  list_add(&kvm->vm_list, &vm_list);

  return kvm;
}
```

结合代码我们看得出，初始化完毕后会将 KVM 加入到一个全局链表头。这样, 我们后面就可以通过这个链表头，遍历所有的 VM 虚拟机了。

创建 vCPU

创建 VM 之后，接下来就是创建我们虚拟机赖以生存的 vCPU 了，代码如下所示。

复制代码

```
virt/kvm/kvm_main.c:
static int kvm_vm_ioctl_create_vcpu(struct kvm *kvm, u32 id)
{
  int r;
  struct kvm_vcpu *vcpu, *v;
    /*调用相关cpu的vcpu_create 通过arch/x86/x86.c 进入vmx.c*/
    vcpu = kvm_arch_vcpu_create(kvm, id);

    /*调用相关cpu的vcpu_setup*/
  r = kvm_arch_vcpu_setup(vcpu);

    /*判断是否达到最大cpu个数*/
  mutex_lock(&kvm->lock);
  if (atomic_read(&kvm->online_vcpus) == KVM_MAX_VCPUS) {
    r = -EINVAL;
```

```
16      goto vcpu_destroy;
17    }
18      kvm->created_vcpus++;
19    mutex_unlock(&kvm->lock);
20
21      /*生成kvm-vcpu控制文件*/
22    /* Now it's all set up, let userspace reach it */
23      kvm_get_kvm(kvm);
24    r = create_vcpu_fd(vcpu);
25
26          kvm_get_kvm(kvm);
27          r = create_vcpu_fd(vcpu);
28          if (r < 0) {
29                  kvm_put_kvm(kvm);
30                  goto unlock_vcpu_destroy;
31          }
32
33          kvm->vcpus[atomic_read(&kvm->online_vcpus)] = vcpu;
34
35          /*
36           * Pairs with smp_rmb() in kvm_get_vcpu.  Write kvm->vcpus
37           * before kvm->online_vcpu's incremented value.
38           */
39          smp_wmb();
40          atomic_inc(&kvm->online_vcpus);
41
42          mutex_unlock(&kvm->lock);
43          kvm_arch_vcpu_postcreate(vcpu);
44
45  }
```

接着，从这部分代码顺藤摸瓜。

我们首先在第 7 行的 kvm_arch_vcpu_create() 函数内进行 vcpu_vmx 结构的申请操作，然后还对 vcpu_vmx 进行了初始化。在这个函数的执行过程中，同时还会设置 CPU 模式寄存器（MSR 寄存器）。

接下来，我们会分别为 guest 和 host 申请页面，并在页面里保存 MSR 寄存器的信息。最后，我们还会申请一个 vmcs 结构，并调用 vmx_vcpu_setup 设置 vCPU 的工作模式，这里就是**实模式**。（一看到把 vCPU 切换回实模式，有没有一种轮回到我们第五节课的感觉？）

vCPU 运行

不过只把 vCPU 创建出来是不够的，我们还要让它运行起来，所以我们来看一下 vcpu_run 函数。

复制代码

```
arch/x86/kvm/x86.c:
static int vcpu_run(struct kvm_vcpu *vcpu)
{
        int r;
        struct kvm *kvm = vcpu->kvm;
        for (;;) {
    /*vcpu进入guest模式*/
                if (kvm_vcpu_running(vcpu)) {
                   r = vcpu_enter_guest(vcpu);
                } else {
                        r = vcpu_block(kvm, vcpu);
                }
                kvm_clear_request(KVM_REQ_PENDING_TIMER, vcpu);

    /*检查是否有阻塞的时钟timer*/
                if (kvm_cpu_has_pending_timer(vcpu))
                        kvm_inject_pending_timer_irqs(vcpu);

    /*检查是否有用户空间的中断注入*/
                if (dm_request_for_irq_injection(vcpu) &&
                        kvm_vcpu_ready_for_interrupt_injection(vcpu)) {
                        r = 0;
                        vcpu->run->exit_reason = KVM_EXIT_IRQ_WINDOW_OPEN;
                        ++vcpu->stat.request_irq_exits;
                        break;
                }
                kvm_check_async_pf_completion(vcpu);

    /*是否有阻塞的signal*/
                if (signal_pending(current)) {
                        r = -EINTR;
                        vcpu->run->exit_reason = KVM_EXIT_INTR;
                        ++vcpu->stat.signal_exits;
                        break;
                }
    /*执行一个调度*/
                 if (need_resched()) {
                         cond_resched();
                 }
          }
```

看到这里，我们终于理解了上文说的 VM-Exit、VM-Entry 指令进入、退出的本质了。这其实是就是通过 vcpu_enter_guest 进入 / 退出 vCPU，在根模式之间来回切换、反复横跳

的过程。

内存虚拟化

在 vcpu 初始化的时候，会调用 kvm_init_mmu 来设置虚拟内存初始化。在这里会有两种不同的模式，一种是基于 EPT 的方式，另一种是基于影子页表实现的 soft mmu 方式。

复制代码

```
arch/x86/kvm/mmu/mmu.c
void kvm_init_mmu(struct kvm_vcpu *vcpu, bool reset_roots)
{
  ......
  /*嵌套虚拟化，我们暂不考虑了 */
        if (mmu_is_nested(vcpu))
                init_kvm_nested_mmu(vcpu);
        else if (tdp_enabled)
                init_kvm_tdp_mmu(vcpu);
        else
                init_kvm_softmmu(vcpu);
}
```

I/O 虚拟化

I/O 虚拟化其实也有两种方案，一种是全虚拟化方案，一种是半虚拟化方案。区别在于全虚拟化会在 VM-exit 退出之后把 IO 交给 QEMU 处理，而半虚拟化则是把 I/O 变成了消息处理，从客户机（guest）机器发消息出来，宿主机（由 host）机器来处理。

复制代码

```
arch/x86/kvm/vmx.c:
static int handle_io(struct kvm_vcpu *vcpu)
{
        unsigned long exit_qualification;
        int size, in, string;
        unsigned port;

        exit_qualification = vmcs_readl(EXIT_QUALIFICATION);
        string = (exit_qualification & 16) != 0;

        ++vcpu->stat.io_exits;

        if (string)
                return kvm_emulate_instruction(vcpu, 0) == EMULATE_DONE;

        port = exit_qualification >> 16;
```

```
17          size = (exit_qualification & 7) + 1;
18          in = (exit_qualification & 8) != 0;
19
20          return kvm_fast_pio(vcpu, size, port, in);
21  }
```

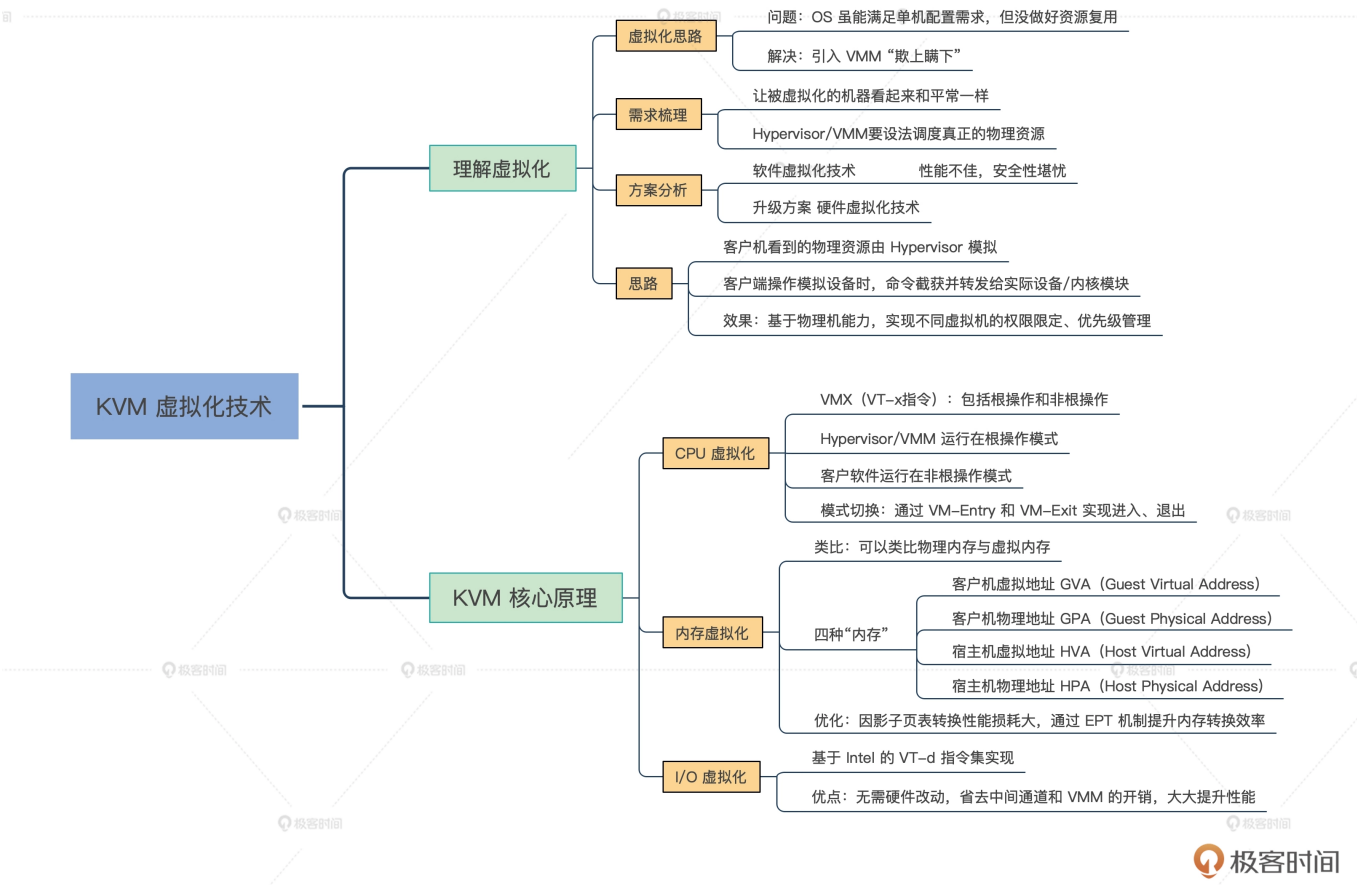
重点回顾

好，这节课的内容告一段落了，我来给你做个总结。历史总是惊人相似，今天我用一个历史故事带你理解了虚拟化的核心思想，引入一个专门的层，像赵高一样瞒天过海，向下统一管理和调度真实的物理资源，向上“骗”虚拟机。

而要想成功实现虚拟化，核心就是对资源进行“欺上瞒下”。我带你梳理分析了 KVM 的基本架构以及 CPU、RAM、I/O 三大件的虚拟化原理。其中，内存虚拟化虽然衍生出了四种内存，但你不妨以用当初物理内存与虚拟内存的思路做类比学习。

之后，我又带你进行了 KVM 核心逻辑相关的代码走读，如果你有兴趣阅读完整的 KVM 代码，可以到官方仓库搜索。

最后，为了帮你巩固今天的学习内容，我特意整理了导图。



思考题

有了 KVM 作为虚拟化的基石之后，如果让你从零开始，设计一款像各大云厂商 IAAS 平台一样的虚拟化平台，还需要考虑哪些问题呢？

欢迎你在留言区跟我互动，也欢迎你把这节课转发给自己的朋友，跟他一起探讨 KVM 的相关问题。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 42 | 瞧一瞧Linux：如何实现系统API？

下一篇 44 | 容器：如何理解容器的实现机制？

更多课程推荐

精选留言 (3)

写留言

pedro 置顶

2021-08-16

以我所在的腾讯云为例。
腾讯云服务器 CVM 支持用户自定义一切资源：CPU、内存、硬盘、网络、安全等，并可以在需求发生变化时轻松地调整它们，还支持随时扩容，迁移，运维等管理功能。
一个成熟的IAAS平台搭建起来不容易啊，要考虑的事情太多了。

展开

作者回复: 如果只是在一台物理机上开启多个虚拟机KVM确实已经做的很棒了，但是如果我们扩展到多个机架、多个机房，问题就变得更加复杂了。我们除了要考虑之前讲过的网络问题，好需要考虑分布式环境下的计算、存储、消息传输、状态同步、动态迁移、扩缩容、镜像、身份认证、编排与调度、UI管理面板等很多问题。当然，业界也有一些开源解决方案，比如大名鼎鼎的OpenStack，不过笔者觉得OpenStack由于设计实现的比较早，所以存在集群规模有限、部署、维护、二次开发复杂度高、历史包袱重等问题，所以和多位架构师沟通交流之后我们正在尝试重新设计并实现一套更现代化的轻量级的IAAS云平台，感兴趣的同学可以加入群多多交流。

1 4

苏流郁宓 置顶
2021-08-16

装虚拟机出现过ghost系统（如win系统ghost版），在VM虚拟机上面识别不了，需要装原版官网下载的非ghost版本才安装成功，以及AMD的cpu装苹果系统也没成功，故认为开发虚拟机系统（如VM），需要对厂家的cpu指令以及各系统的启动流程有一定的了解。如此才好“欺骗”，虚拟机安装系统失败的很大部分原因是没找对原版系统，也就是修改过的系统启动流程和虚拟机表中的流程不完全一致，识别不了。...
展开

作者回复: 66666

1 1

springXu
2021-08-16

这课的例子和内容相当精彩，还是意犹未尽呀。由于虚拟化知识欠缺了，想问还有后续不？

作者回复: 暂时没有 可能加餐